AMP Virtio
V0.5

Bill Mills (bill.mills@linaro.org)
with contributions from many

Introduction

Context

Terms:

AMP virtio-mmio	A variation of virtio-mmio that works in AMP mode. It can work without a shared hypervisor or without blocking VM-exits or Hypervisor request calls (like HVM)
AMP Bank	A data structure that describes a set of virtio-mmio resources. The AMP Bank data structure is normally placed at the start of a memory segment that is shared between two or more peers. It describes the AMP virtio-mmio areas, which peer is the driver side and which peer is the device side. It also describes areas of memory shared between the peers. AMP Bank is optional. It is not required for AMP virtio-mmio if the needed data is provided in some other way.
Device side	The OS/software that provides the virtual device
Driver side	The OS/software the has the driver and uses the virtual device
Peer	Either the Device side or the Driver side. Also “ours” and “theirs” is used to refer to “my” data and the other peer’s data.
Coordinator	Any software that provides a coordination function to the device and driver sides. This can be as simple as a bootloader that fills a section of memory with 0’s or can be more complex like a VMM that provides device hot plug and dynamic shared memory allocation
Shared Memory	A section of memory that is accessible by both the Device side and the Driver side. This can be on chip memory or DDR or both.
Shared Notification	The Device side and Driver side each have a method to send a notification to the other. This notification is out of band from the mmio memory structure and is typically a mailbox or doorbell mechanism. The notification should cause an IRQ on the peer. This notification may be shared with other uses: it may serve multiple virtio devices and other unrelated activities.
Unique Notification	Unique notification provides an indication that a specific event has occurred at least once. The event will be a specific transport configuration event or an event associated with a specific virtqueue. The virtio-mmio,amp memory structure will have facilities to identify unique notification events when signaled from a shared notification. If a mailbox mechanism can transport 32 bits of data, then it can provide both the shared notification and unique notification for multiple contexts. In this case the virtio-mmio,amp notification memory structures are optional and their use may be configured away.

Platform Examples

Typical AMP SOC

Many SOCs designed for industrial and automotive applications have multiple CPU core types on the SOC. Many of these SOCs feature multiple Arm A profiles cpus in an SMP cluster and then add co-processors cores (Cortex-M/R, DSP, etc) for various other system integrator defined tasks.

Examples include:

- AMD Xilinx ZynqMP and Versal SOCs (Cortex-A + Cortex-R [+ FPGA softcores])
- NXP iMX 8* (Cortex-A + Cortex-M [+ DSP])
- ST MP1* (Cortex-A + Cortex-M)
- TI AM64* (Cortex-A + Cortex-R)

PCI based AMP

AMP systems can be constructed by connecting different systems with PCIe or other memory addressing bus.

PCIe root complex (Host) and PCIe end-point (card)

A simple AMP PCIe system can be created by adding a PCIe card with an Arm based SOC to an x86 based host. The Arm system in this case acts as a PCIe endpoint [EP] and the x86 host is the PCIe root complex (RC).

Such a system normally has MSI[-X] for EP to RC notification and some MMR on the EP for RC to EP notification. The EP normally will share some portion of its memory with the RC. The areas accessible to the RC are normally configurable via some EP internal method. The RC will also normally allow the EP to access some portion of its memory but this is not strictly required for the system to operate. If the RC makes some memory accessible to the EP it will normally use an IOMMU to control what is allowed.

This system is AMP in multiple ways.

- The CPU architectures are different
- One side or the other might not be Linux
- Different OS instances are present w/o a common hypervisor
    - So even if the Host and Card CPUs are Arm and both running Linux, it is still AMP

Two PCIe root complex systems connected via a non-transparent bridge

Another type of PCIe based AMP system is similar to the above but both systems are root-complex (RC) from the PCIe view. These systems are then connected by a special type of PCIe switch called an non-transparent bridge.

This system ends up looking very similar to the RC + EP system but the EP specific portions are replaced with bridge specific portions. The bridge will have some way for each side to generate MSI[-X] notifications to the other and will each configure sections of memory that will be accessible via BAR access to the other. An IOMMU is not strictly needed in this case if you trust the bridge as the bridge will only allow access to the memory regions that you programmed into the bridge.

Xen system with RTOS and Linux guests in a Mixed critical system

In this system, the whole system is running on an SMP capable cluster, for example 8 Cortex-A CPUs. The Xen hypervisor is being used and each CPU shares the same Xen hypervisor. The advantage to using AMP virtio in this case might not be immediately obvious. However the system may:

- Have a mix of RTOS and Linux guests
- Have a mix of critical needs for the guests
    - Real-time latency requirements
    - Security requirements
    - Safety certification levels
- A desire to use Virtio based devices (instead of Xen specific PV devices)

Xen today supports virtio using the existing virtio-mmio protocol. Each read or write access to the virtio-mmio region stops the vCPU and sends a IOREQ message to the device model for that device. (Typically all devices for a given guest domain (DOMU) are served by an instance of QEMU in DOM0 but other configurations are possible.) The vCPU is halted and can not make progress until the device model signals that the IOREQ is complete.

In a mixed critical system the hypervisor will need to meet the real-time, security, safety and availability requirements of the union of its guests. However not every guest can or should meet that union of requirements. It can be difficult and unnecessary to have Linux or the GPU driver meet all of those requirements.

Using AMP virtio, the hypervisor requirements are reduced and the device side is decoupled from the driver side. A realtime or safety critical driver side can still use a device served from a domain that does not meet these requirements. The real-time performance of the critical guest is never impeded and the safety analysis does not need to include the device model and its host OS. The critical guest must be prepared for the device to be slow or become unusable but it remains in control of its own operation.

A typical configuration for Xen will use pre-shared memory and Xen events. Xen events do use a hypervisor call but execution is returned quickly to the vCPU and thus the worst case latency can be defined.

Test Platforms

To ease development the following development platforms are used for this effort.

Dual QEMU w/ IVSHMEM

Two independent QEMU systems are used. Each is connected to an IVSHMEM device that provides a shared memory and a method for signaling other peers. Normally the IVSHMEM device appears as a PCI device and can be used with QEMU models like arm64 virt model and x86 models. In addition there are patches to QEMU to enable a non-pci based IVSHMEM device to be connected to a Cortex-M model. The other side of this device can be any IVSHMEM client but is typically the PCI based client.

Each QEMU system is free to run its own OS.

This allows a variety of systems to be constructed:

- Cortex-A to Cortex-A
- Cortex-A to Cortex-M
- x86_64 to Cortex-A
- etc

ZynqMP QEMU

AMD Xilinx has a version of QEMU that can run a ZynqMP SOC model. This model has four A53s and two R5s.

QEMU Arm64 running Xen

For the Xen use cases a single QEMU instance is used and Xen is run with a variety of guest OSes.

Out of Scope systems

Systems that are only connected via non-memory based connections are **not in scope** for this work. Examples include two systems connected via ethernet or a host and a co-processor connected via serial port, I2C, or SPI. (LPC bus or eSPI could be in scope but are not currently being evaluated.)

Solution Summary

The following is a summary of the issues and the proposed solutions.

1. Delay of driver side initialization
    a. Early boot code or a VMM should fill the MMIO area with zeros before either side is active
    b. The driver side should delay using the MMIO area until the MAGIC value, Device version number, and Virtio Subsystem Device ID are valid
    c. The device side should only populate these fields once it is ready for activity. (It may populate some early but it should hold off at least one.)
    d. The driver side should monitor the Magic/Version/Device ID fields as well as the Status field to help detect asynchronous device side reset
    e. If the driver side discovers an already operational state in the MMIO area on fresh reboot, it should issue a reset to the device.

2. AMP mode recognition
    a. The driver side may know that the device is in AMP mode from DTB or other means
    b. Regardless a new version number will be used for AMP that both indicates AMP mode and allows new layouts in the future
    c. It is assumed that the device knows it is to operate in AMP mode

3. Notification method
    a. A system level notification between the peers must be provided but does not need to be fine grain or unique.
    b. A new in-band fine grained notification structure is defined in the MMIO memory that provides 64 notifications from device to driver and 64 notifications from driver to device
        i. 16 notifications in each direction are allocated for transport usage
        ii. The remaining notifications are per virtqueue allowing up to 48 virtqueues
        iii. Each notification has an event level acknowledgement independent of semantic acknowledgement
    c. If the system level notifications can provide all the fine grained notifications it may be used instead of the MMIO in-band notifications. Such usage must be pre-configured on both sides.

4. Index based configuration
    a. The device must insure the data associated with the current index value is valid before it signals it is ready for activity via clause 1.c

b. When the driver side changes a configuration index (feature index, virtqueue index, or shared memory segment index) it will send a transport level notification as defined above and wait for the corresponding index update done notification
c. The driver is then free to read the data associated with the index
d. If the driver changes the data associated with the index it will send a data changed notification and wait for its explicit data change done notification. Each type of index has it own set of data change events

5. Configuration changes
    a. The device must populate the config data and set the config generation to 0 before signaling it is ready for activity.
    b. Before any change to the config data, the device must increment the config generation (which will make it odd). After the change is done the device will increment the config generation again making it even. The device will then send a config change notification.
    c. When reading the config data, the driver should read the config generation, read the data, and read the config generation again. If the two values are not the same or are odd, the process should restart. (Reading the data can be skipped if the generation is odd but the above procedure makes it look similar to the currently required method.)
    d. If the driver is going to change the config data it SHOULD wait if the config generation is odd
    e. If the driver is going to change the config data it SHOULD read the config generation before and after the config data change and retry the operation if not the same. (This is a SHOULD as it may be hard to fit into existing virtio frameworks.)
    f. After the driver has changed the config data it will send a config changed notification to the device.
    g. When the device receives the config change notification it will need to find the changed data.
    h. The device SHOULD NOT poll the config data for changes or read it to affect its current operation. It is expected any important data is cached in local data structures on the device side.

6. Device needs to know PA space of driver
    a. Immediate solution: Device needs to be aware of the driver side PA address of each shared memory segment. This is the current solution to be compatible with the Linux kernel.

        In this model the device must use the driver side PA values for virtqueues and buffer addresses and will need to translate them unless DRV_PA == DEV_PA

b. Better solution: All addresses between the peers are expressed as a memory segment index encoded into the top address bits and an offset into that segment in the lower address bits

7. Virtqueue and buffer allocation
    a. In MOST uses of AMP virtio-mmio, virtqueue data structures and buffers must be constrained to pre-shared memory areas. AMP virtio-mmio may find use when the device can access any portion of the drive side memory. In such uses, the constraints in this section (7) do not apply
    b. The driver side must be aware of the shared memory segments.
    c. The driver must allocate virtqueues and buffers within these memory segments
    d. Memory segment location and size may be discovered from DTB, AMP Bank, xenstore. etc

8. Memory coherency issues
    a. As with all cases of virtio, memory barriers must be used to insure any changes are visible to the peer
    b. If the peers are cache coherent, cached memory may be used for MMIO, virtqueues, and buffers.
        i. In this case atomic operations (LoadLink/StoreCondition) can also be employed
    c. If the peers are not cache coherent, then
        i. uncached memory should be used for the MMIO region to avoid false sharing issues
        ii. It is recommended to use SRAM for the MMIO region if available as uncached accesses to DDR are very inefficient
        iii. cached memory may be used for virtqueue and buffer areas but cache invalidate & flush operations will need to be used
    d. coprocessors that are io-coherent with the host processor will allow manual cache ops to be skipped on the host OS
        i. The coprocessor may then used non-cached mapping or cached mapping with manual cache operations
    e. Except as noted above (8.b.i) atomic operations are unlikely to work between peers. They likely will not work even between threads of one peer if using uncached memory. A multithreaded peer should protect multiple access to the MMIO area (for example) from its multiple threads by other means such as a mutex. Use of atomic ops between the peers should not be relied upon.

Transport Related Issues

Delay of Driver Side Initialization

In classic virtio the driver side exists and is ready to respond before the driver side starts. For AMP virtio this is not always true. We need a way to signal the driver side not to interact with the device until it is ready.

An early coordinator must fill any possible virtio-mmio transport areas (Normally 0x200 bytes) with zeros before either side starts. As AMP virtio normally uses memory carve outs, filling the carve out with zeros is sufficient. The early coordinator does not need to know the exact layout that will be used by each virtio peer.

The Driver side should ignore any virtio-mmio area where the Magic value, Device version number, or the Virtio Subsystem Device ID is 0. The driver should recheck the 3 fields on a slow poll or when the device sends a shared notification event. (Either time based poll or wait for notification will work. It is the driver side choice of which to use.).

The device side should be 100% ready and all other fields should be set before it sets the Virtio Subsystem Device ID. When it does set this field, it should send a shared notification event to the driver side.

Peer Reboot or Crash

Asynchronous reset / reboot / crash of the other side is also an issue that needs to be dealt with. If the other side just stops responding that is an issue that needs to be handled via timeouts or out of band means. If the other side executes a crash handler steps can be taken to notify the other side. In any case once the other side is operational again re-enumeration of the device is required.

These measures include **at least** the following procedures.

If the driver side detects the status field resets to 0 unexpectedly, it should immediately stop using the device and tear down any device usage inside itself. No reset is possible in this state as the device is already quiescent. This same procedure applies if there is any change in the Magic, Version, or DeviceType fields but in this case the Status field should also be 0.

On the driver side, if the device issues a reset request but never acknowledges the reset command after a generous timeout, the driver side should assume a crash handler set the reset request bit but was not around to handle the reset itself. In this case the driver should assume the device is in a similar state as the status field asynchronously being set to 0.

A crash handler on the driver side ***should*** send a reset but it is not required to wait for the response.

On the device side after a fresh boot, if the magic/version/device type are valid and the state is not 0, the device should request a device reset. If the device type is to change the device type should be set to 0 before the reset request is sent. After the device has been reset by the driver side and acknowledged by the device, it can change the device type and signal the driver side as a hint that it can re-evaluate the fields.

On the device side, a crash handler ***should*** set the reset request bit but does not need to wait for a response.

On the device side, if it detects a timeout or is notified that the driver side has crashed, the device side should stop writing to the vrings and buffers and set the reset request bit.

AMP Mode Recognition

AMP virtio uses a variant of virtio-mmio transport. However this variant is not 100% compatible with the non-amp aware mode of virtio-mmio. The driver side needs to know if it should use the traditional mode or the amp aware mode. The device side needs to populate the virtio-mmio fields so it must know its mode when it is initialized.

In all likelihood, the driver side will also already know if it should work in AMP mode. The AMP mode requires an out of band notification method such as a doorbell register and IRQ or a bi-directional mailbox. These facilities will vary by platform and so no attempt is made to describe them in the MMIO structure. For a devicetree based platform, the location of AMP virtio mmio structures will be described in the DTB along with the notification method. **The DTB should use a different compatible string for virtio-mmio AMP mode**. Thus the driver side should already know if AMP mode is required.

Identification of needed material for a non-DT based platform is TBD for now. It could be included as subdevices of an already supported driver. An example would be a PCI device driver that defined a bi-directional notification method and used the AMP Bank structure to define the layout of the virtio-mmio areas.

Regardless of the above, it seems best to mark the virtio-mmio areas in a way that will provide an added check. Possabilities include:

- Changing the magic number
- Changing the version / layout number

Here we suggest **changing the version/layout number to 0x0001_0003.** The 3 indicates a new layout version and the 0x0001_XXXX is the first layout option field and states that AMP mode is required.

The features bits may seem like a good candidate for this functionality but it is not. Because of the way feature negotiation is defined, the driver will need to know if AMP mode is required before it does feature negotiation.

Notification Method

The notification methods defined in virtio-mmio (and virtio-pci) are not usable in the AMP mode. They rely on hypervisor magic (trap and emulate) memory for the virtio-mmio region. The notification from driver to device is a magic memory write and can identify individual virtqueue and can optionally supply extra data to the device. The device to driver notification is via an IRQ but demultiplexed based on a MMIO register. The device to driver notification only differentiates config change vs a ganged virtqueue notification. The driver must check each virtqueue when for work if the ganged notification is signaled.

AMP mode requires new notifications to solve the Index Based Configuration issues and to signal driver to device config space changes.

AMP mode requires a shared notification mechanism. The platform capabilities of that shared notification will vary greatly. In the minimal case a single shared notification should work for a group of AMP virtio-mmio devices. This minimal case and others will require unique notifications to be provided by the specific virtio-mmio data structure. The section below describes this unique notification mechanism.

In-band unique notification:

This proposal defines a method to define 64 device to driver notifications and 64 driver to device notifications within the virtio-mmio data structure. The structure chosen uses two memory areas. One memory area is only written by the driver and only read by the device. The other is only written by the device and only read by the driver. Each unique notification uses 1 bit in “our” word for signaling and 1 corresponding bit in “their” word for acknowledgement. Therefore each 64 bit word in the region enables 64 notifications. The first 16 notifications will be reserved for the transport level notifications leaving 48 unique virtqueue notifications. A device or driver may choose to use 32 bit reads and writes instead of 64 bit but that is expected to be slightly less efficient. If using 32 bit words and there are less than 16 virtqueues, the second set of words does not need to be read or updated.

The inband notification does not support extra data delivered as part of the Driver to Device notification so this must not be offered by the device when in-band notification is used.

The device writable words will be at offsets 0x60 to 0x6F replacing the InterruptStatus, InterruptACK and two more unused 32 bit words.

The driver writable words will be at offsets 0xD0 to 0XDF. These words are currently unallocated in the Virtio 1.2 specification. This offset is chosen to put the two regions in different cache lines if the cache line is 128 bytes or less. A max cache line size of 256 would be better but that would change the size of virtio-mmio space and move the offset of the device specific

config space. Uncached memory will need to be used for the whole MMIO if the shared cache line is 256 bytes or larger and the peers are not coherent. Due to the nature of the config data area, uncached memory is recommended anyway for non coherent peers.

Offset	Writable	Function	Bits 31-16	Bits 15 to 0
0x60	Device	Device to Driver Events	Notifications 31 to 16 (virtqueues 15 to 0)	Notifications 15 to 0 (for Transport)
0x64	Device	Device to Driver Events	Notifications 63 to 48 (virtqueues 47 to 32)	Notifications 47 to 32 (virtqueues 31 to 16)
0x68	Device	Driver to Device Event ACKs	Notifications 31 to 16 (virtqueues 15 to 0)	Notifications 15 to 0 (for Transport)
0x6C	Device	Driver to Device Event ACKs	Notifications 63 to 48 (virtqueues 47 to 32)	Notifications 47 to 32 (virtqueues 31 to 16)
0xD0	Driver	Driver to Device Events	Notifications 31 to 16 (virtqueues 15 to 0)	Notifications 15 to 0 (for Transport)
0xD4	Driver	Driver to Device Events	Notifications 63 to 48 (virtqueues 47 to 32)	Notifications 47 to 32 (virtqueues 31 to 16)
0xD8	Driver	Device to Driver Event ACKs	Notifications 31 to 16 (virtqueues 15 to 0)	Notifications 15 to 0 (for Transport)
0xDC	Driver	Device to Driver Events ACKs	Notifications 63 to 48 (virtqueues 47 to 32)	Notifications 47 to 32 (virtqueues 31 to 16)

Notification #	Driver to Device	Device to Driver
0	General Driver status field or config space change	General Device to Driver config space change
1	Feature Index Updated	Feature Index Update Complete
2	Feature Info Updated	Feature Info Update Complete
3	Queue Index Updated	Queue Index Updated Complete
4	Queue Info Updated	Queue Info Update Complete
5	SM Index Updated	SM Index Update Complete

6	SM Info Updated (reserved)	SM Info Update complete (reserved)
7	TBD	Reset Requested
8 to 12	Reserved for future common transport use	Reserved for future common transport use
12 to 15	Reserved for Protocol specific use	Reserved for Protocol specific use
16	Buffers available virtqueue 0*	Return of Buffers virtqueue 0*
…	“”	“”
63	Buffers available virtqueue 47*	Return of Buffers virtqueue 47*

Out of band unique notification:

The out of band notification is normally some sort of doorbell or cross interrupt mechanism.

In most configurations the above unique notification structure will be used to carry the notifications for the virtio connection and the out of band notification will be used to activate it. Exceptions are possible and will be described below.

On AMP SOCs this is normally called a mailbox. However the capabilities of a mailbox vary a lot. Some are capable of delivering small (16 to 64 byte) messages, some deliver a scalar value (commonly 32 bit), and others just deliver an event notification. For mailboxes that carry data, the depth of the mailbox (how many unread messages can be sent at one time) varies greatly also. Some have a depth of 1 while others may allow 4 or even 16.

Mailboxes that carry data can use a scheme to carry the unique notifications directly. It is up to the system integrator to decide to do that or not. Note: when mailbox data is used for unique notification, there is no event level ack. Even when mailboxes carry data it may be best to use the unique notification structure above to reduce pressure on the mailbox depth. To use mailbox data for unique notification, a message like below could be used:

31 to 24	23 to 16	15 to 8	7 to 0
Virtio MMIO MAGIC or message ID		MMIO instance	Unique ID

Use of such mailbox-passed notification must be known to both sides before virtio negotiation.

On PCI based systems the notification will normally appear as an MSI[-X] toward the host (RC) and MMR for the host to poke. For end-points the representation will vary. In any case the

notification is often just a notification without data at least on the host side. The in-band notifications are expected to be used on all PCI based systems.

For Xen based systems the shared notification will normally be the Xen events system provided by the hypervisor. This is Xen specific code but is minimal and easy to support and already supported in Zephyr. Xen events carry no data. There can be a large number of events but managing such events for each unique notification above would be awkward so it is expected that one event be used in each direction per AMP virtio and the in-band notifications will be used.

Index Based Configuration

Virtio-mmio uses index based configuration for features, virtqueues, and shared memory segments. The normal usage is for the driver to write an index, and then read and maybe write the data associated with this data. It may be that the hypervisor & device will modify the associated data based on the date value written. Such effects will also appear instantaneously to the driver.

This system relies on the hypervisor intercepting the write to index, holding off the vCPU, and updating the data associated with the index. After the guest resumes (immediately after the index write operation), it can rely on the data being updated. Likewise the hypervisor will detect changes to the data by intercepting the write to these data areas and can update associated data if needed.

For AMP virtio-mmio we can not rely on such interceptions. Instead updates will be coordinated with unique notifications. The driver will:

1. Write the index value
2. Send the appropriate index update event
3. Wait for the index update done event
4. Read the associated data
5. If data is to be modified
    a. modify one or more associated data values
    b. send the appropriate data update event
    c. wait of the data update done event
6. [Re-read data if the driver is allowed to change it based on writes]

Normally one data update cycle is all that will be required but the sequence in #5 & #6 above can be repeated if desired.

Configuration Space

The configuration space in virtio is normally setup by the device before communication begins to describe the device. Some updates from the device at runtime are possible and required by some protocols. In virtio 1.0 and later a generation counter is provided by the device to help the driver side detect config changes. Likewise, the driver side can update the config data in some

situations. The protocols are defined such that simultaneous updates from the device and driver to the same memory location are not expected.

The above situation is not hard to support and is really all that is required in most real usage of virtio with existing protocols. The above usage is what the spec actually recommends.

However, the spec ALLOWS several things that make the support of the config space harder:

1. Writes and even reads to the config space can cause instantaneous side-effects
2. Any read and write side effects are defined at the protocol level not the transport level
3. Simultaneous writes from both sides to the same location are ill-defined
4. The driver may read the config data asynchronous while the device is changing it

Numbers 1 to 3 only work today because the hypervisor intercepts reads and writes to the config space and allows the device to act on them and potentially make corresponding updates to the virtio mmio area. Number 2 above prevents a system from fully implementing virtio-transport in the hypervisor and leaving protocol level to the device model.

The generation counter was added to virtio in the first standard version (1.0) to avoid corner cases in the driver reading config data while the driver is updating it (Number 4 above). Such updates cause conflict when the data read is larger than a single atomic scaler (32 or 64 etc based on the specifics of the system). The classic example of this given in the spec is the mac address of a virtio-net device. As this is a 48 bit value, the driver could read the old first 32 bit value and the new value of the last 16 bits. To avoid this situation the driver is supposed to read the config generation counter before and after the config data reads and retry the operation if the two generation counters readings are not the same. Even this generation counter solution relies on the hypervisor intercepting reads from the driver side to ensure updates to the data and updates to the generation counter are atomic.

For AMP Virtio-mmio, there really are no solutions to #1 and #2. Number 3 could be worked around by having the driver read the config generation before and after any write to the config data and redo the operation if they are not the same. For #4 the generation counter could be made to work with a new semantic definition such that odd config generation counter values indicate an ongoing update from the device side.

See the solution summary for the config data procedures.

Memory Related Issues

Device needs to know PA space of Driver

In traditional virtio the device side knows about the physical memory map of the driver side. It also can access any portion of it.

Newer virtio methods are adding access control and explicit sharing to the driver side so that it does an explicit share operation to allow the device to access memory. This model is much like a real hardware device whose accesses are controlled by an IOMMU. This model is implemented by virtio-iommu and by the Xen page grant mechanism among others. In this model the device side gets a physical address and range for each shared area and is thus informed of the PA for each accessible portion of memory.

Driver side physical addresses (DRV_PA) are used to define the virtqueue areas and are used for the buffer pointers in the virtqueues. In virtio 1.2, optional shared memory segments were added for use by the protocols. These shared memory segments are known to the device side and reported to the driver side using driver side physical addresses as well.

The use of DRV_PA represents an issue for AMP virtio where the device side may only have limited knowledge of the driver side PA layout. There are two solutions proposed to address this issue: pre-configuration and segmented memory references.

Device pre-configuration of DRV_PA

With this model, nothing changes from the driver side point of view. DRV_PA will continue to be used. To make this work the device side will need more information about each memory segment that may be used in virtio.

There are of course some simple cases. On many AMP SOCs, the device side physical addresses (DEV_PA) will be the same as DRV_PA. This is common when Linux is running without a hypervisor and communicating with a co-processor. However if the Linux instance is running under a hypervisor, the PA seen by the Linux guest (IPA for Arm) will normally not be the same as true PA. In this case the device will need to use the IPA of the guest. This might be known ahead of time but might not be known until the Linux guest is running.

Solutions for this issue include:

1. DTB or other pre-configured boot data when DRV_PA to DEV_PA relationship is known ahead of time.
2. A rendezvous memory structure in shared memory where each peer can report information about themselves. Such a structure is envisioned here and called AMP Bank but details are yet to be finalized.
3. When dynamic memory sharing is used, a hypervisor or other coordinator will be involved. The coordinator will know both the DRV_PA and the real PA and could be setup to share this information with the device side. This is easy if the device side is also a guest of the hypervisor but could be done via hypervisor to device side message if the device side is not a guest of the same hypervisor.

Segmented memory references

With this model each DRV_PA is replaced with a segmented memory reference constructed with the memory segment ID in the top bits of the scalar value and the offset within the segment

at the lower bits of scalar value. How many bits to allocate to each portion can be debated. The offset should have at least 22 bits (4MB segments) and the memory segment ID should have at least 6 bits (64 segments.) As DRV_PAs today are 64 bit, the suggested split would be 12 bits for segment ID (4096 segments) and 52 bits for offset (4096 terabytes). It is recommended that segment ID value of zero is reserved to ensure any accidental use of DRV_PA values is detected.

Use of segmented memory references will need to be negotiated as a transport level feature bit.

Virtqueue and Buffer Allocation

(See solution summary)

Memory Coherency Issues

(See solution summary)

AMP Bank memory structure

The AMP Bank memory structure is an optional add-on to the AMP virtio-mmio definition. AMP virtio-mmio can be used without it. An AMP SOC use case can use DTB on each side (or System DT) to describe the information needed.

However the AMP Bank memory structure solves some of the issues of pre-configuration of both sides that are hard to do with more loosely coupled systems like PCIe based AMP or Xen. In these cases the needed information is not known until run-time and then only visible to the one side. The data structure below allows each side to populate its information into a structure that can be probed by the other side.

It may also solve issues in connecting ACPI based systems.

It could also be used to implement hot-plug/unplug by an active coordinator.

A typical use of AMP Bank would have only 2 peers but could have 3 if the coordinator is an active role. However it is possible to have more peers in a single AMP Bank structure if that makes sense from the system point of view. Of course a given peer may participate in more than one AMP Bank system with each having a different set of peers. The peer and memory segment IDs used in multiple AMP Bank systems should be coherent if there is any shared visibility.

The exact structure of the AMP back data structures is TBD but the below should be valid statements.

The AMP Bank memory structure has the following properties:

1. Consists of multiple segments
    a. each segment is writable by a single entity.
    b. each segment is at least 256 bytes but can be expanded to 4K or 64K if it is desired to enforce memory access with an MMU
        i. This extra space can be used for other structures that share the same protection needs
    c. At least one segment will be writable by the coordinator
    d. At least one segment will be writable by each peer
2. The main coordinator memory structure will define
    a. 0 to N preconfigured memory areas for use with virtio (buffers vrings etc)
        i. It is expected that the coordinator structure will be in one of the memory segments and it be owned by the coordinator
    b. 0 to N peer information segments
    c. 0 to N AMP virtio-mmio areas
3. Memory area definitions will contain
    a. A magic number to mark the segment as an AMP Bank memory segment
    b. A system unique ID (UUID or system significant ordinal number)
    c. Size of the area

    d. Location hints, examples include:
        i. system unified bus address
        ii. VID:PID, Serial Number, Bar ID, offset
        iii. Offset from other memory segment
    e. Allocation owner Peer index
        i. The allocation owner is in charge of all allocations in the memory area
4. The Peer information area will contain
    a. A magic number to identity the structure
    b. A system unique ID for the peer
    c. A memory segment record for each memory segment it can see
        i. The record contains the memory segment ID and the PA of that segment from the POV of the peer
    d. A notification record for each other peer this peer may communicate with
        i. This records contains the other peers ID and data about notification that the other peer will need to know
        ii. Examples include:
            1. Xen: My dom ID & Event ID to send
            2. IVSHMEM My peer Id and event
            3. MSI info
5. The AMP virtio records will contain
    a. The Peer ID of the device side
    b. The Peer ID of the driver side
    c. The memory segment and offset of the AMP Virtio structure
    d. Allowed memory segments for use by vrings, buffers and shm
    e. (Driver always owns allocation of vrings and buffers)
    f. Today in Virtio 1.2, it is believed by this author that the device side always owns allocation in the shm